INDEX

LABORATÓRIO DE BANCO DE DADOS

VANESSA BORGES

Apresentação baseada em:

- Ramez Elmasri e Shamkant B. Navathe 6° Ed (2010)
  - Capítulo 18: Estruturas de indexação para arquivos

Definição de índices

- A ideia por trás de um índice ordenado é semelhante à que está por trás do **índice usado em um livro**, que lista termos importantes ao final, em ordem alfabética, junto com uma lista dos números de página onde o termo aparece no livro.

- Podemos pesquisar o índice do livro em busca de certo termo em seu interior e encontrar uma lista de *endereços* — números de página, nesse caso — e usar esses endereços para localizar as páginas especificadas primeiro e depois *procurar o* termo em cada página citada.

Índices – conceitos básicos

- Ás vezes, é necessário recuperar os registros de uma tabela especificando os valores de um ou mais campos
- Exemplo:
  - Encontrar todos os estudantes cujo curso é ‘Ciência da Computação’
  - Encontrar todos os estudantes cujo número do departamento > 7

- **Um índice recupera os registros de forma rápida por meio de uma chave de pesquisa**
  - Qualquer subconjunto dos campos de uma relação podem ser uma chave de pesquisa
  - **Chave de pesquisa não é necessariamente uma chave primária**

Índices

Entrada no arquivo de índice

1
2
3
4
5
6
7
8
9
10
11
12

Entrada no arquivo de dados

4	Marta	F	123456789
6	Maria Ferreira	F	896589658
10	João Constantino	M	256462589
1	Pedro Villela	M	0909877878
5	Iza Vargas	F	7890098776
8	Luiz Carlos Albu	M	2343135556
2	Marisa C. Vilas	F	345453454
11	Patrícia	F	123567899
9	Joaquim Lincohn	M	000987654
3	Luzia M. Costa	F	125896325
12	Vargas de Souza	M	365896987
7	Cláudia M. Britto	F	225698969

Índices

- **É armazenado fisicamente**
- Possui apenas um ponteiro para o posicionamento físico da tupla no disco
- Não contém todos os atributos da tupla, apenas os indexados
- **Deve existir um índice para cada chave primária**

Os tipos mais predominantes de índices são baseados em arquivos ordenados (índices de único nível) e estruturas de dados em árvore (índices multinível, B-trees).

Índices ordenados (único nível)

- **Uma estrutura de índice** é criada para cada arquivo no disco
- Utiliza-se um **atributo específico** para gerar as entradas na estrutura de índice
- O atributo escolhido chama-se **atributo de indexação**
- O índice armazena, para cada um dos valores do **campo de indexação**, todos os endereços de bloco onde o campo de indexação aparece (como índice de livro)
- Os valores dos índices são ordenados (Busca Binária)

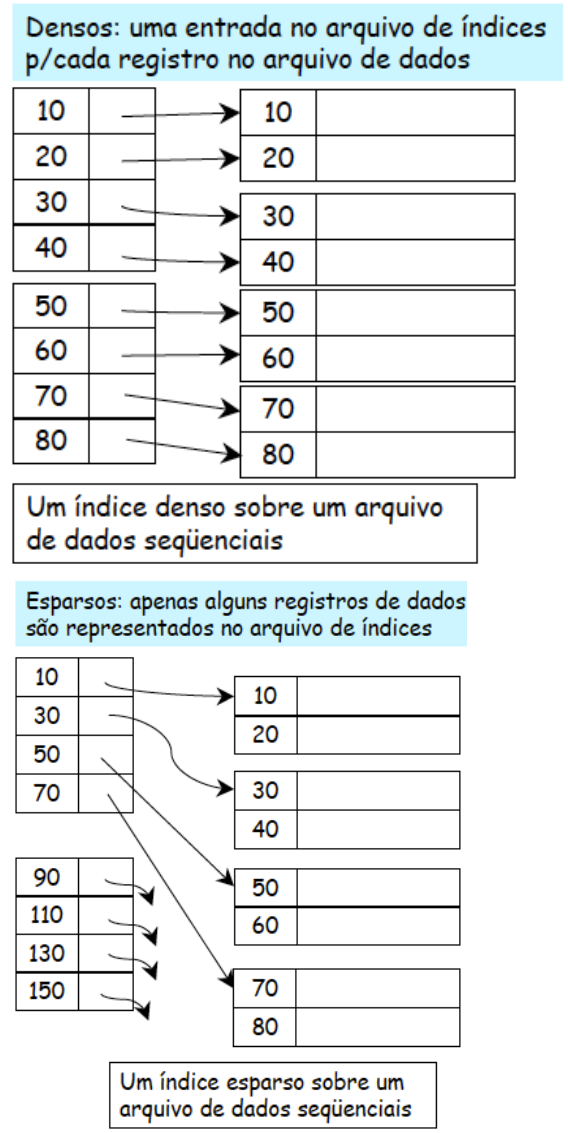
- **Classificação:**
  - Denso ou Esparso

- **Tipos:**
  - Primário
  - Agrupados (*Clustered*)
  - Secundário

Índices – Classificação

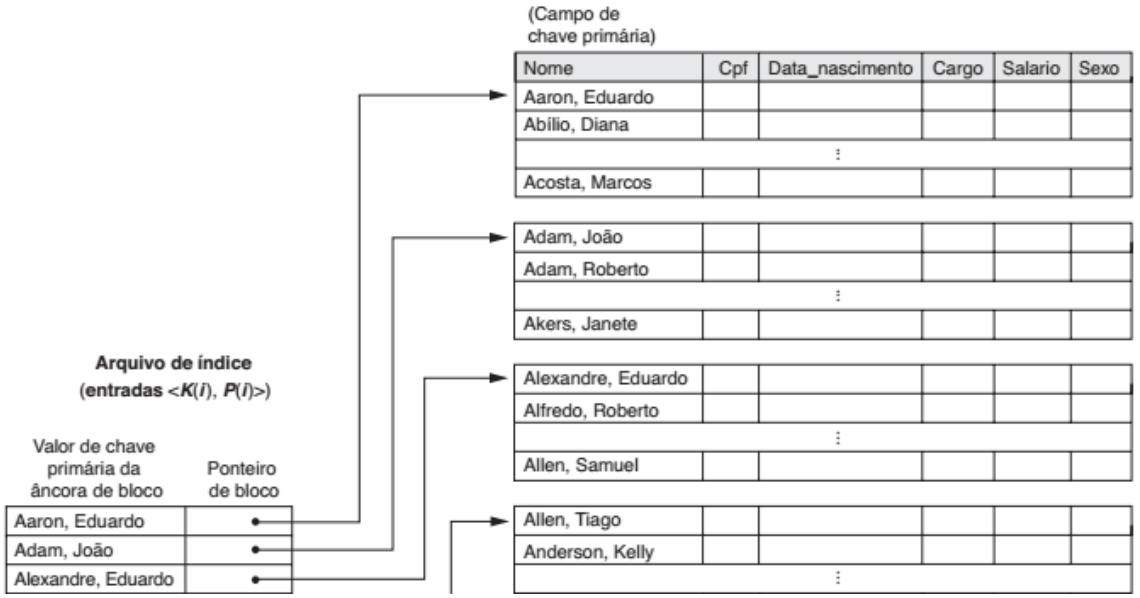
- **Denso**
  - Uma entrada no arquivo de índice para cada registro no arquivo de dados
  - Sequência de blocos contendo apenas as **chaves** dos registros e os ponteiros para os próprios registros
  - Possui menos entradas do número de registros no arquivo

- **Esparso**
  - Uma entrada no arquivo de índice **para cada página de arquivo de dados**
  - Usa menos espaço de armazenamento, porém leva mais tempo para localizar um registro dada a sua chave



Índice ordenado – Primário

- **Primário**
  - **É baseado na chave primária**
  - Segue a mesma ordem do arquivo de dados
  - Aponta para blocos de registros (**é esparso**)
  - O **primeiro campo** é do mesmo tipo de dado do campo de **chave de ordenação**
    - O **segundo campo** é um ponteiro para um bloco de disco (um endereço de bloco)
  - O **número total de entradas** no índice é igual ao **número de blocos de disco** no arquivo de dados ordenado.
  - **Só pode ser criado se o arquivo de dados foi ordenado pelo atributo chave**



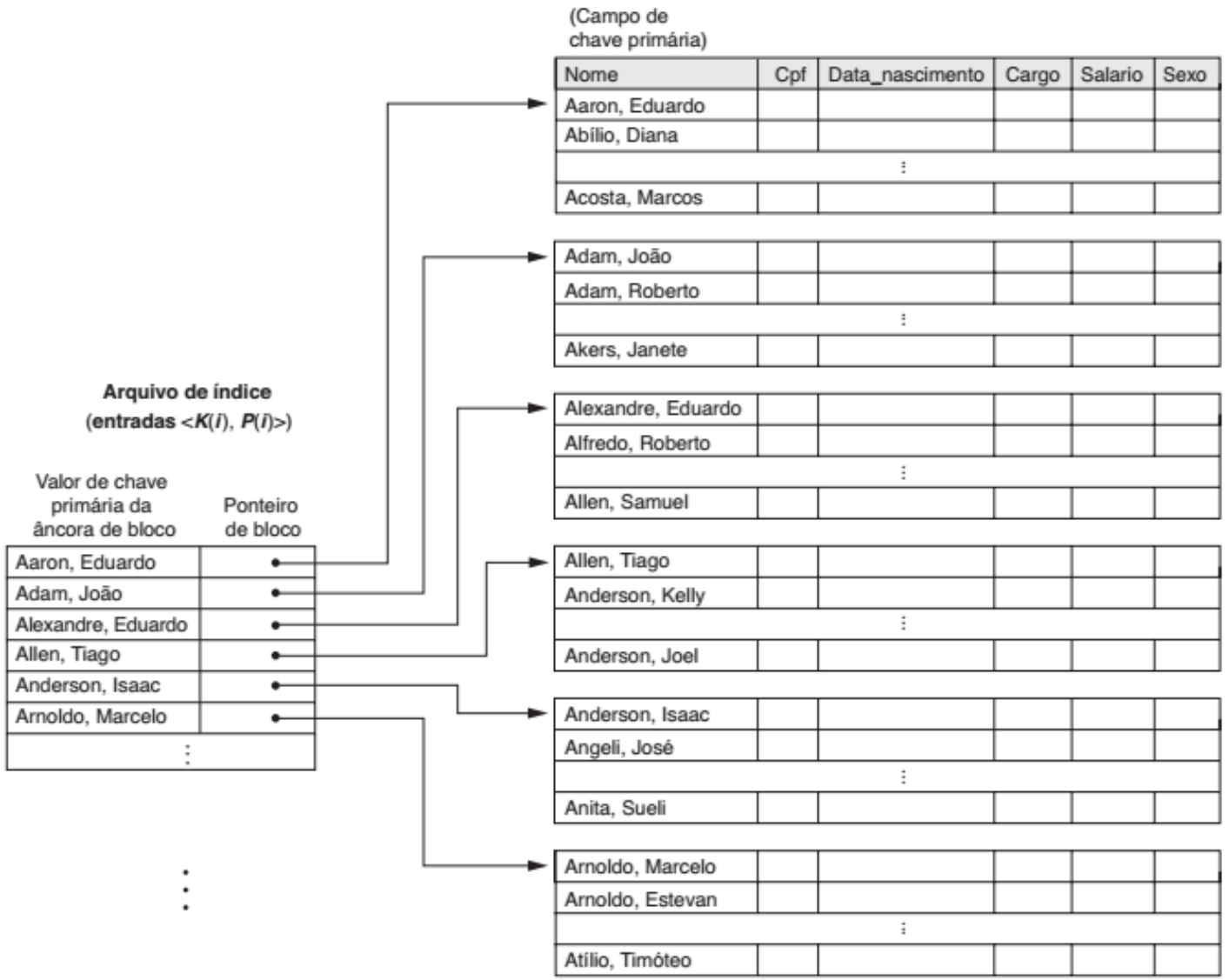
- O índice é um arquivo ordenado com registros do tipo **<K(i), P(i)>**, onde:
  - **K(i)** é o valor do campo chave primária para o **primeiro** registro no bloco i
  - **P(i)** ponteiro para o bloco i

Índice ordenado – Primário

- O arquivo de dados já está armazenado no disco, só então o arquivo de índice é criado....
- **Construção do arquivo de índice:**
  - Percorre-se todos os blocos que armazenam os arquivos
  - Em cada bloco i
    - Pega-se o primeiro registro armazenado (chamado de âncora do bloco)
    - Com base no atributo chave de classificação, cria-se a entrada P(i), K(i), onde P(i) é o valor do atributo chave (de classificação) para o registro âncora e K(i) é o endereço do bloco i

Arquivo de índice é menor que o arquivo de dados
→ há menos entradas
→ apenas dois campos (cada entrada)

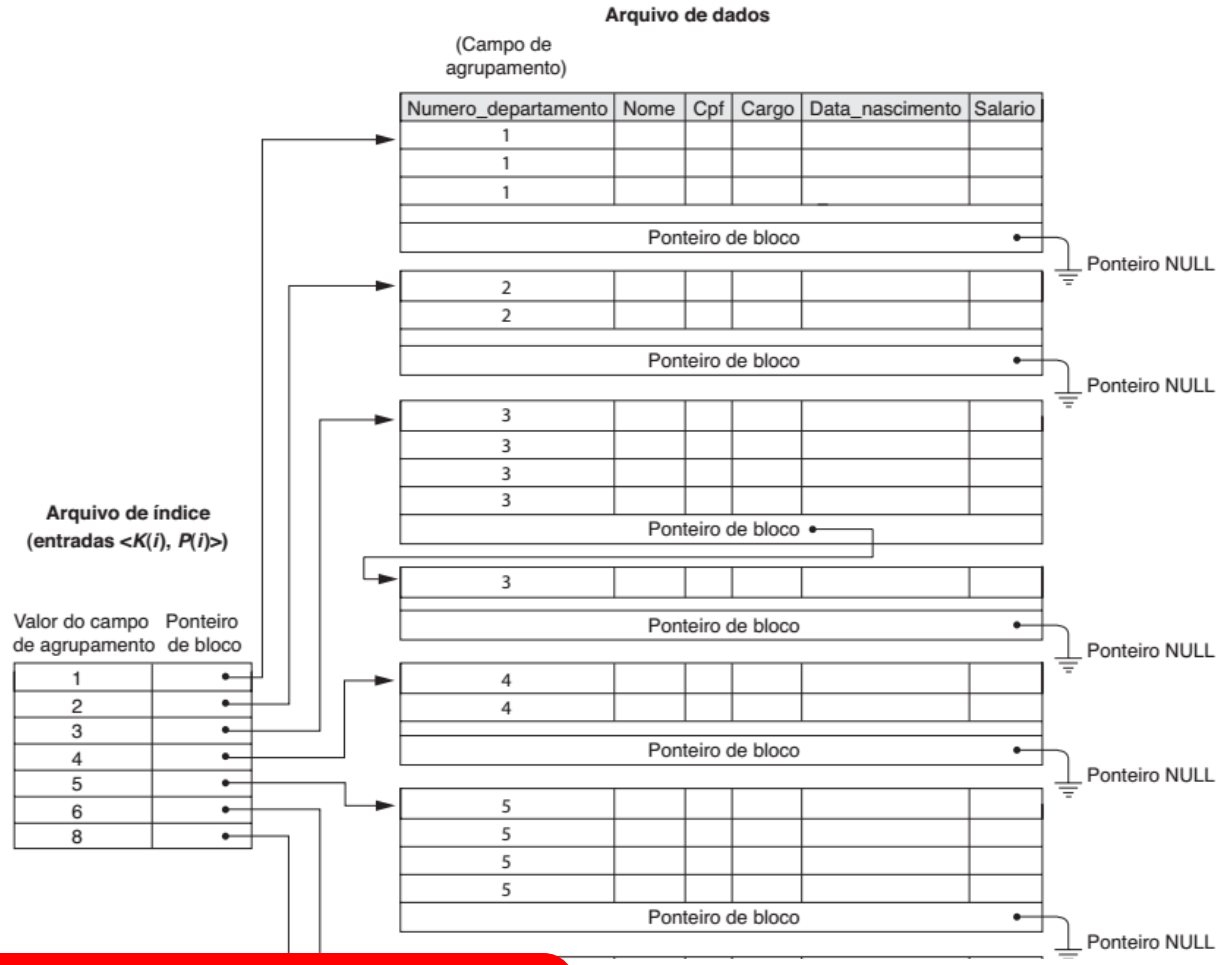
Busca binária mais eficiente



Índice ordenado – Agrupados

- Criado para arquivo de **registros ordenado por atributo não-chave** (campo de agrupamento)
- Existe **uma entrada no arquivo de índice para cada valor distinto** do campo de agrupamento
- Aponta para blocos de registros (**é esparso**)

  - É um arquivo ordenado com registros do tipo **<K(i), P(i)>**, onde:
    - **K(i)** é o valor do campo de agrupamento i
    - **P(i)** ponteiro para o primeiro bloco que armazena registros com determinado valor para o campo de cluster i

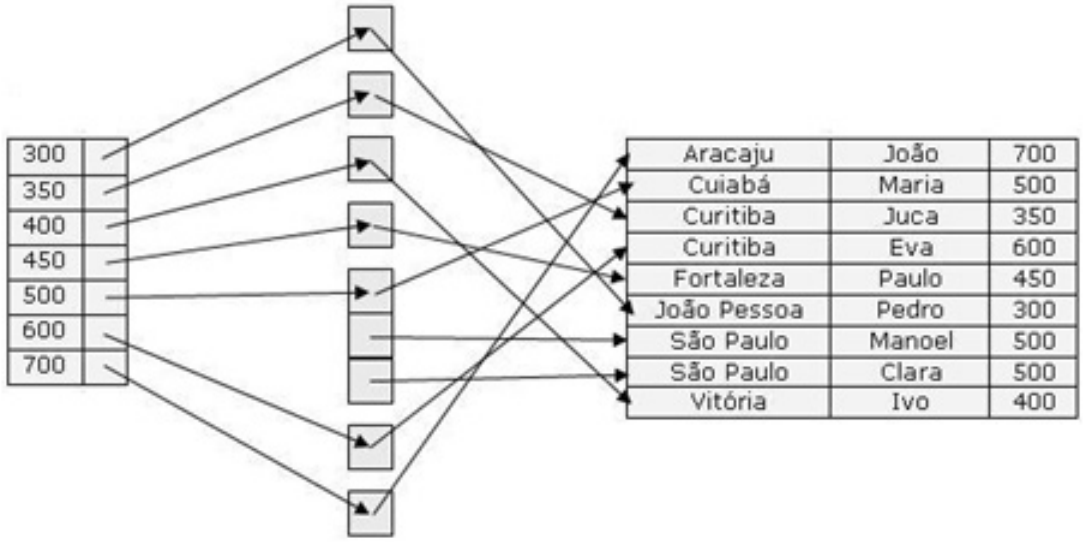


A inclusão e a exclusão podem causar problemas, visto que o arquivo está ordenado.

É preciso reorganizar os ponteiro do arquivo de índice, bem como os dados nos blocos do disco

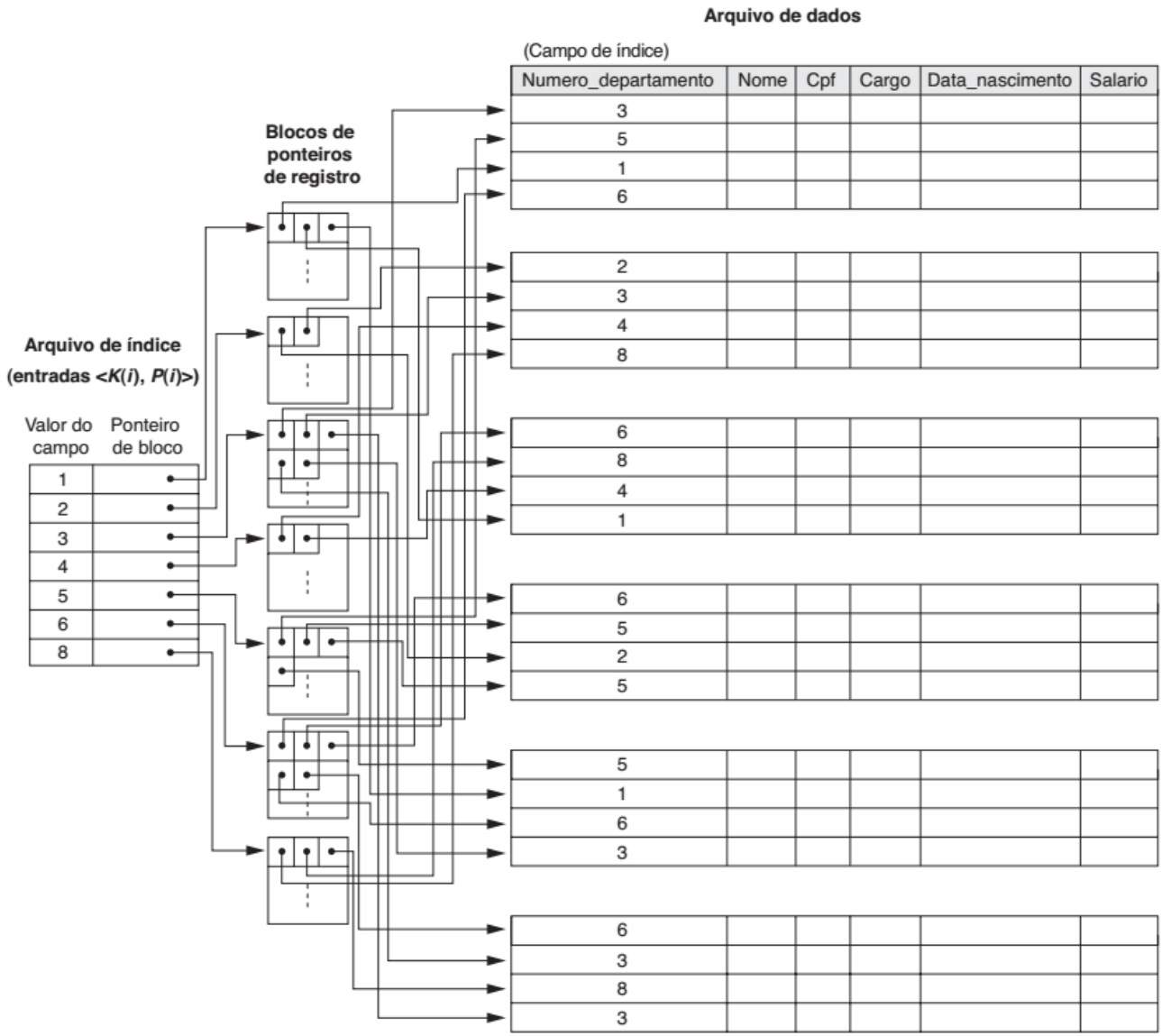
Índice ordenado – Secundário

- **Secundário**
  - Um índice secundário oferece um meio secundário para acessar um arquivo de dados para o qual **algum acesso primário já existe**
  - Os registros do arquivo **poderiam ser ordenados ou desordenados**
  - Pode ser criado em **campos chave ou não chave**
  - **Índice denso**
  - Os ponteiros **não apontam diretamente para o arquivo com registros**, mas para um **bucket** que contém ponteiros para o arquivo



Índice ordenado – Secundário

- É criado **outro arquivo de índices** para armazenar os dados do próprio arquivo de índice.
- Neste esquema **esparso**, o ponteiro P(i) na entrada de índice <K(i), P(i)> aponta para um **bloco de ponteiros de registros**.
- Cada ponteiro de registro, naquele bloco, aponta para um dos registro do arquivo de dados com valor K(i)



Índices Multiníveis

- Um índice multinível é um “Índice de índice”.
  - **Primeiro nível:** arquivo ordenado pela chave de indexação, valores distintos, entradas de tamanho fixo.
  - **Demais níveis:** índice primário sobre o índice do nível anterior e assim sucessivamente até que no último nível o índice ocupe apenas um bloco.
  - **Número de acessos a bloco:** um a cada nível de índice, mais um ao bloco do arquivo de dados.

- **Motivação:** se o arquivo de índices se torna muito grande para ser armazenado em bloco de disco, é interessante indexá-lo em mais de um nível

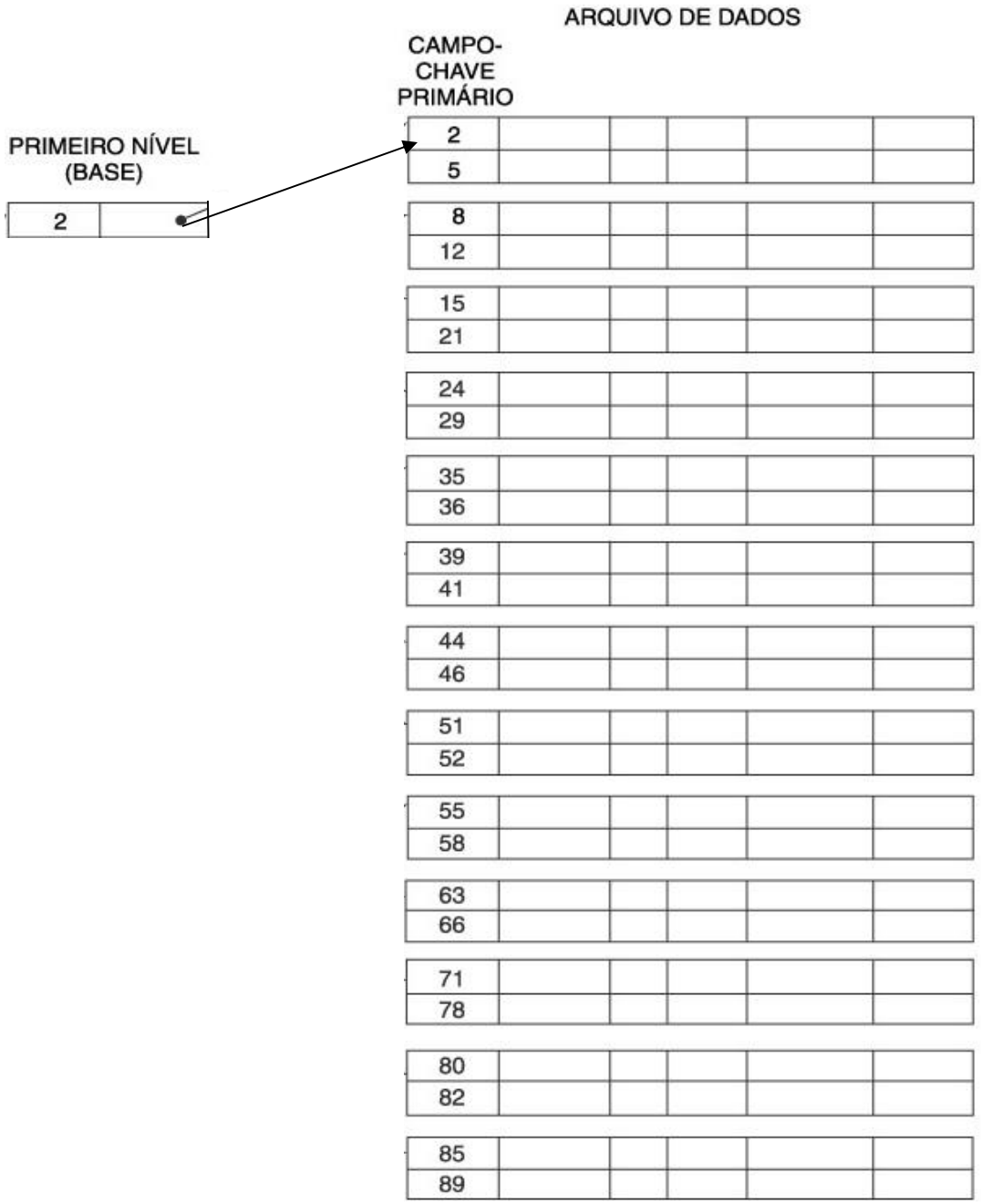
- **Vantagem:** índice pequeno pode ser mantido em memória e o tempo de busca é mais baixo

Construindo o 1º nível (Índice Primário)

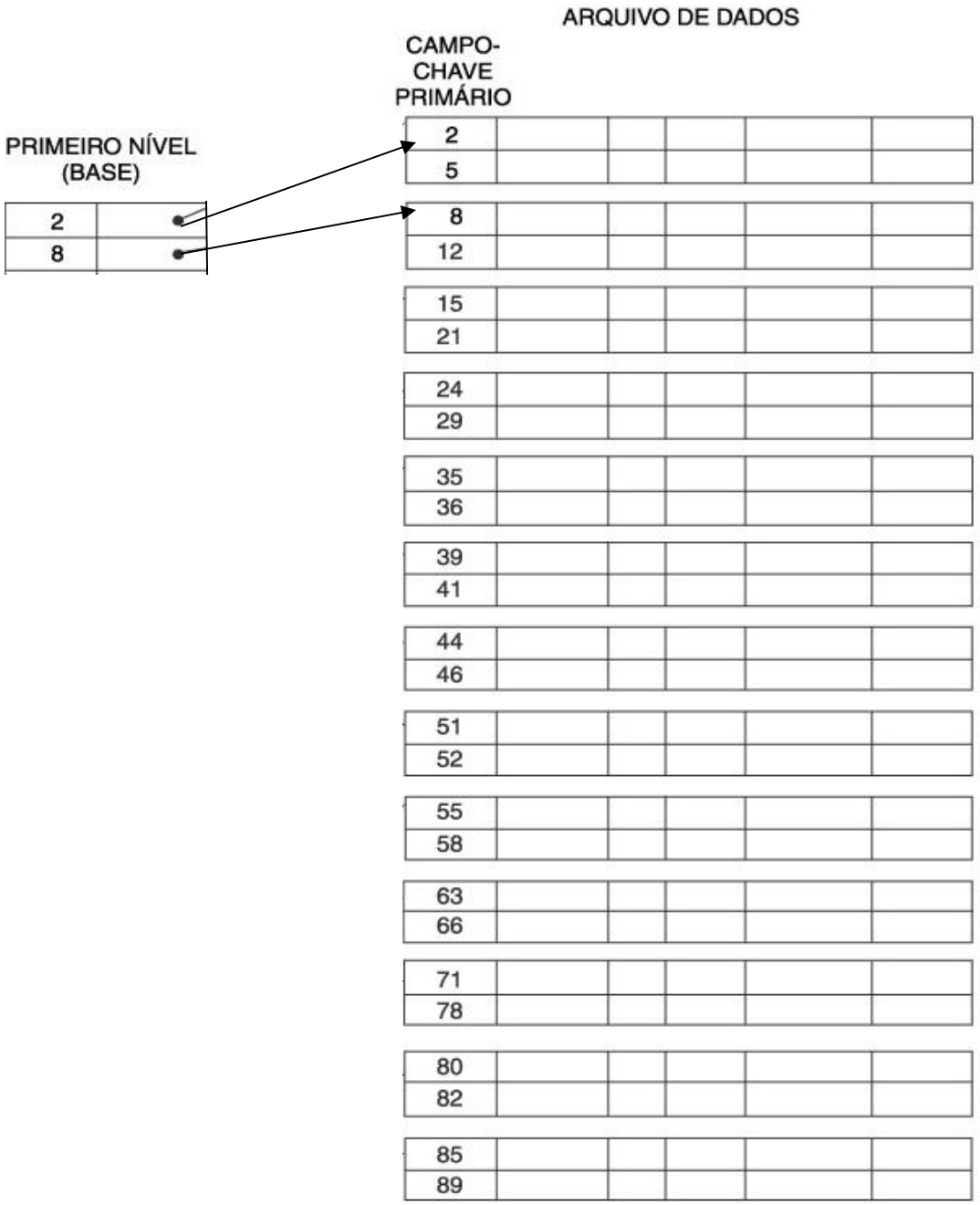
ARQUIVO DE DADOS

CAMPO-CHAVE PRIMÁRIO					
2					
5					
8					
12					
15					
21					
24					
29					
35					
36					
39					
41					
44					
46					
51					
52					
55					
58					
63					
66					
71					
78					
80					
82					
85					
89					

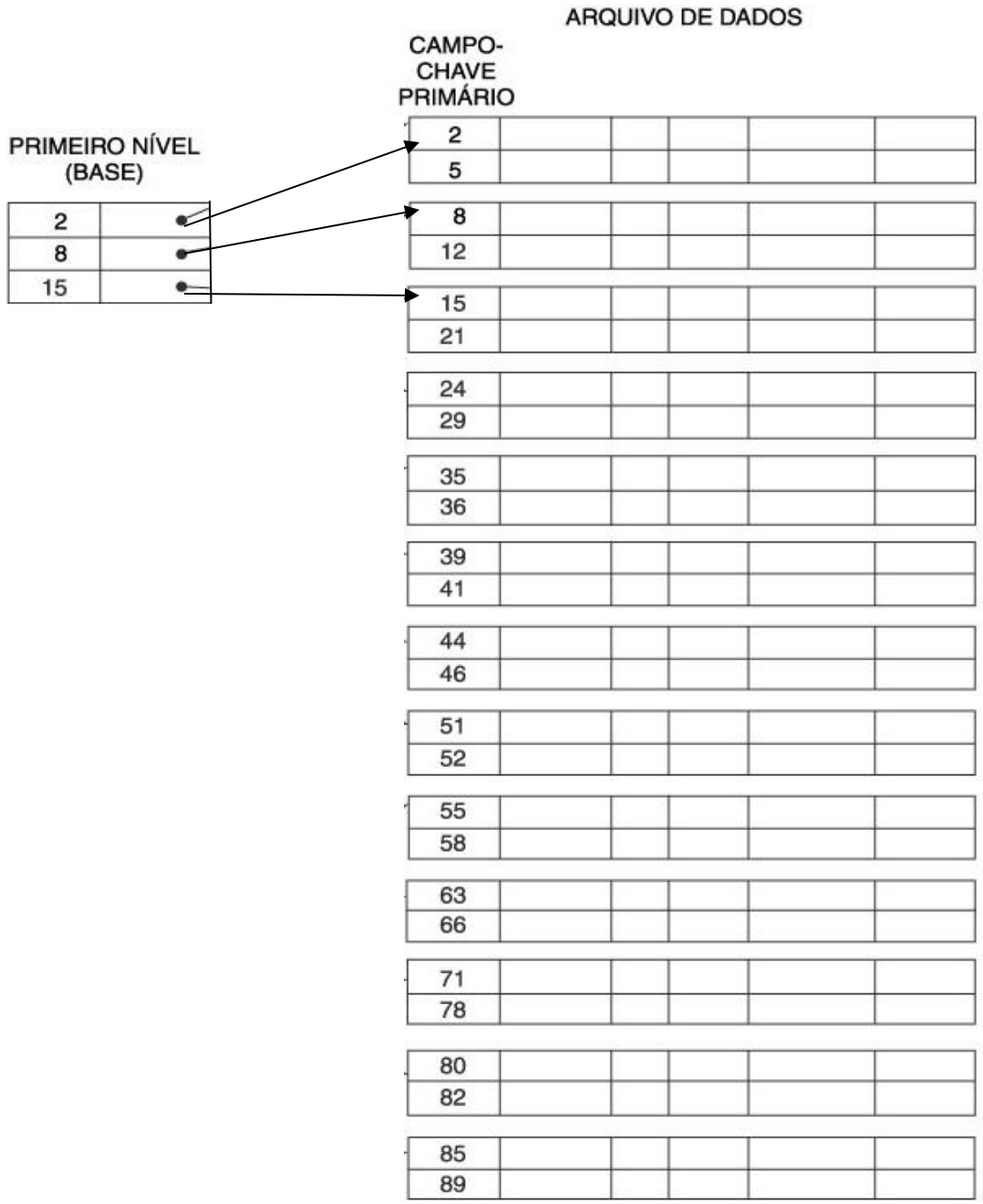
Construindo o 1º nível (Índice Primário)



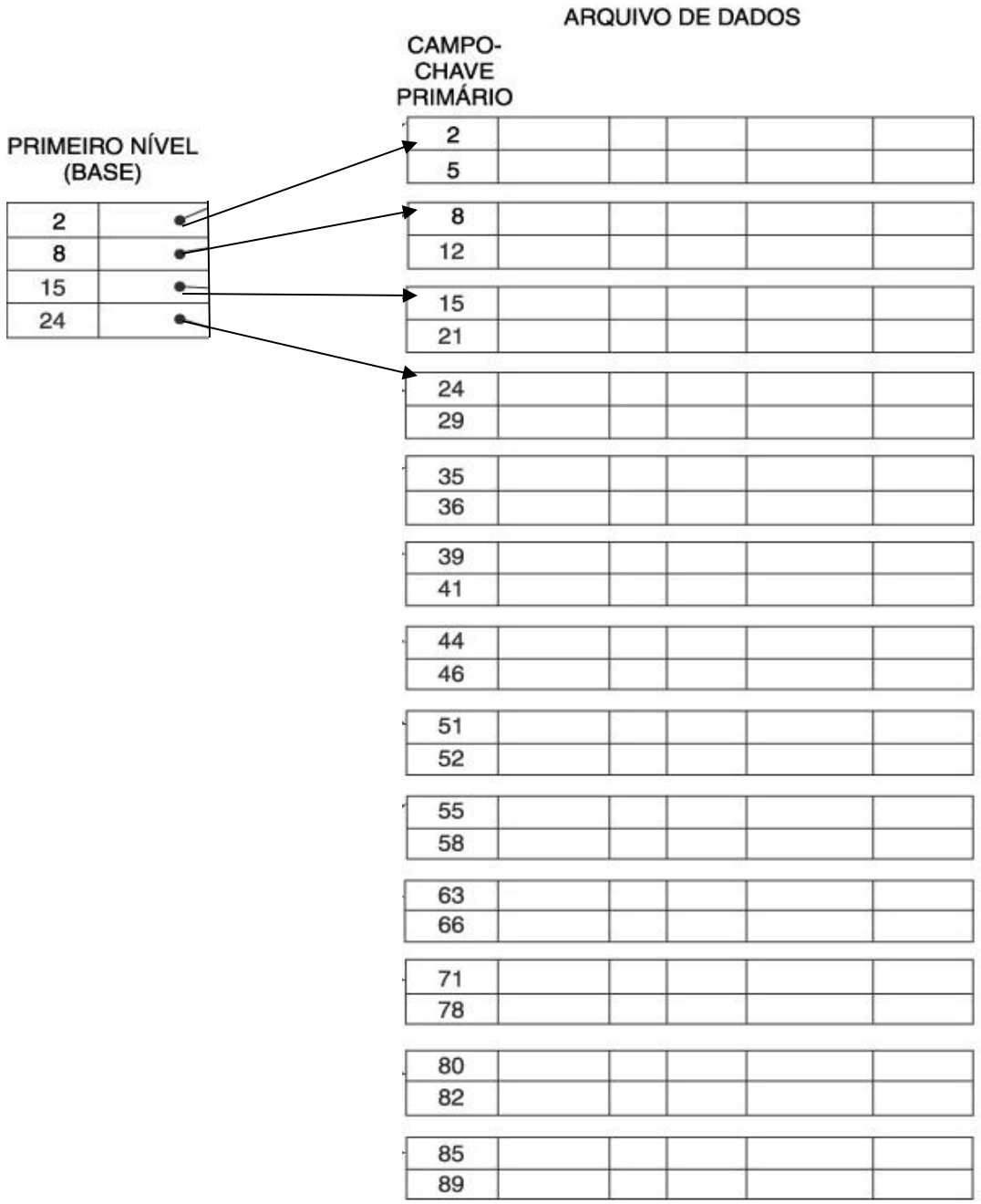
Construindo o 1º nível (Índice Primário)



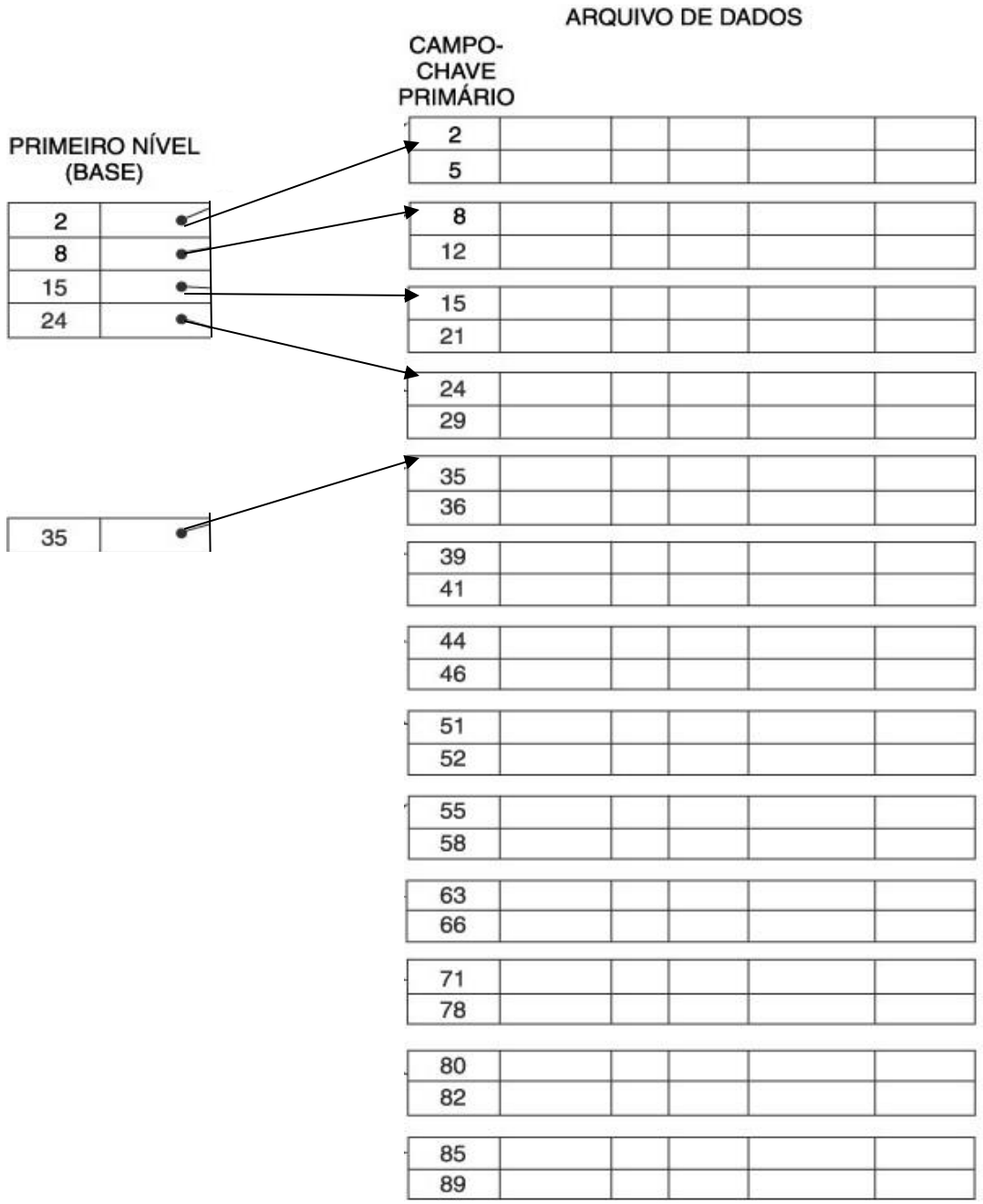
Construindo o 1º nível (Índice Primário)



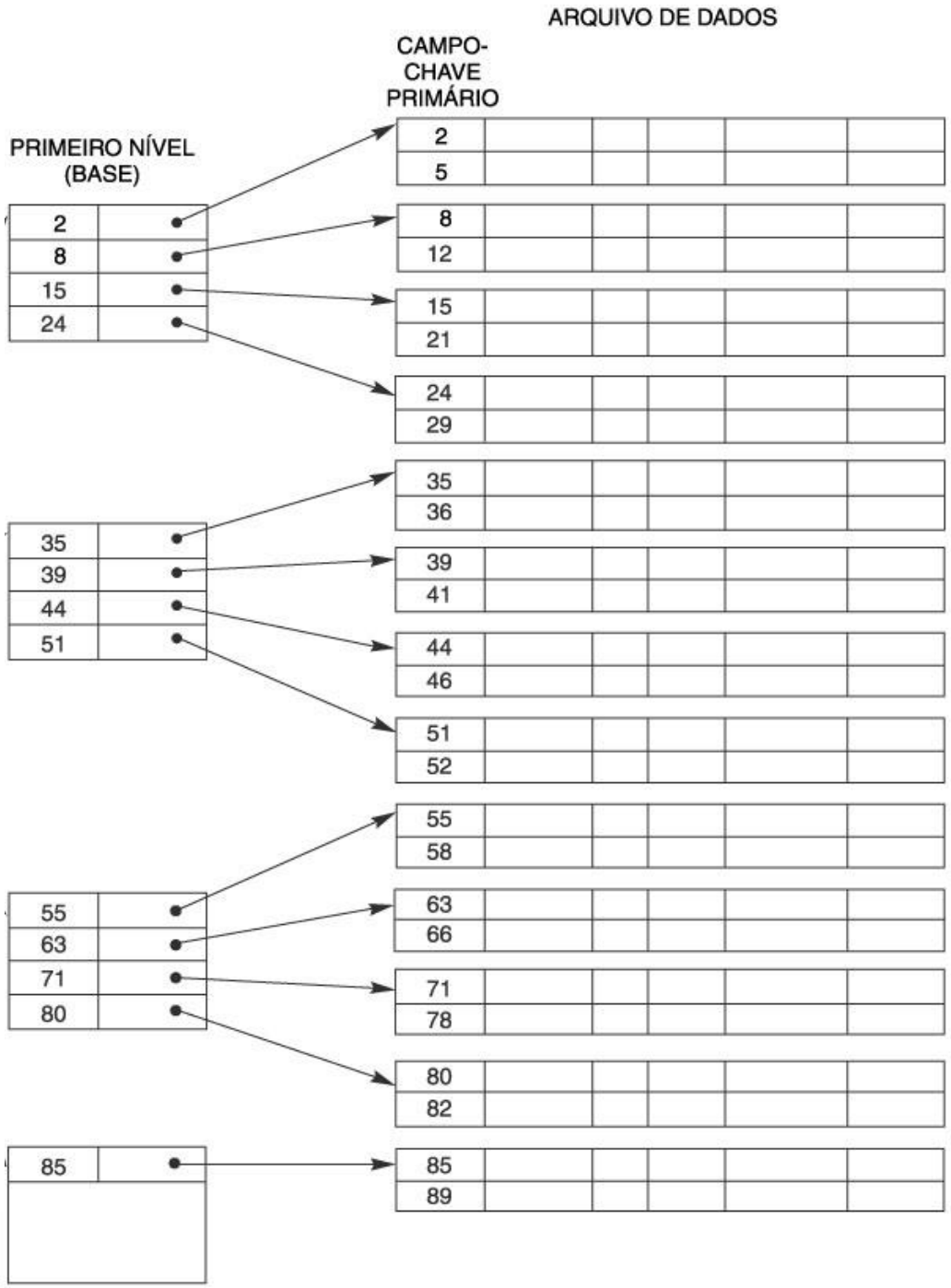
Construindo o 1º nível (Índice Primário)



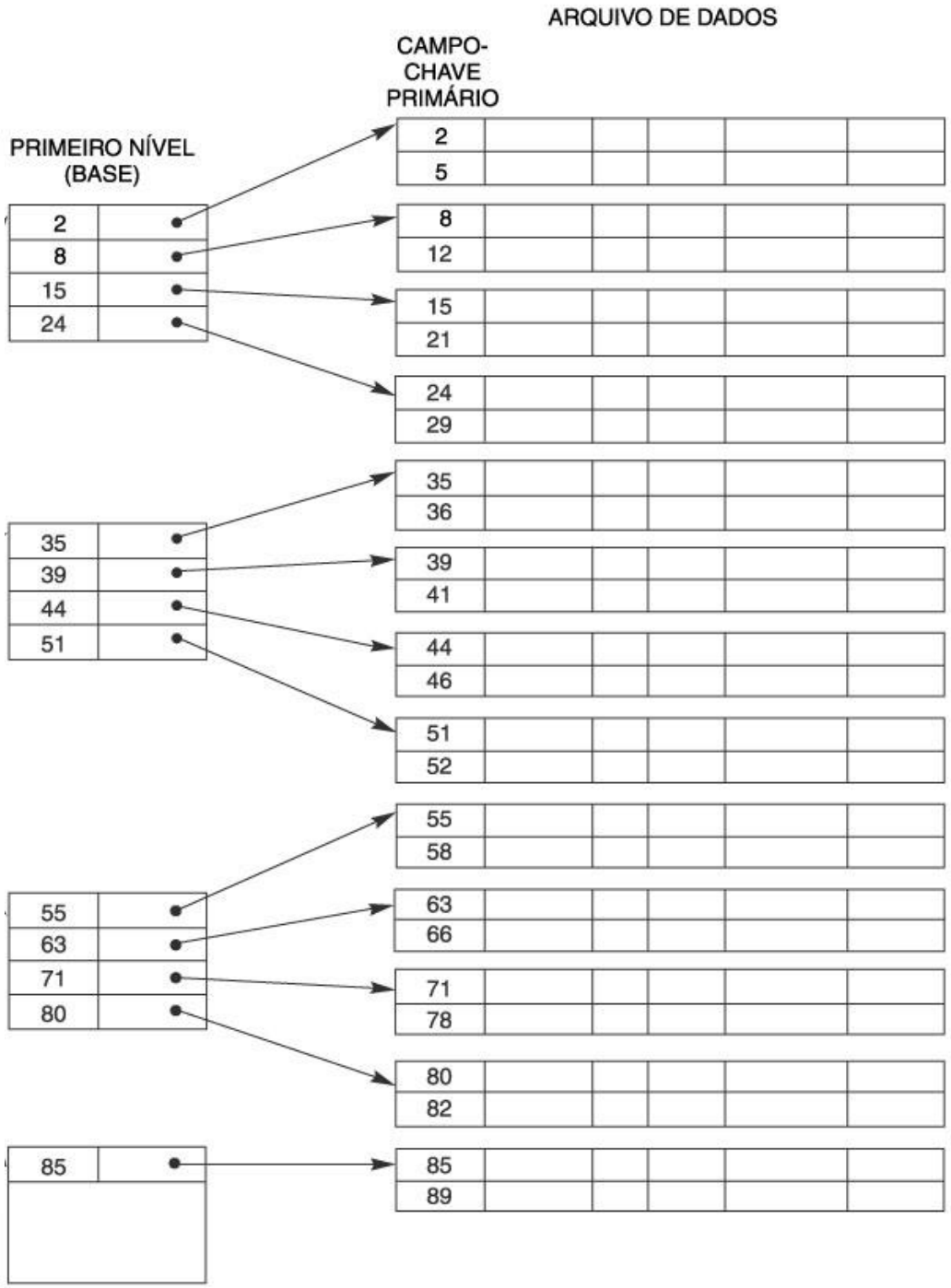
Construindo o 1º nível (Índice Primário)



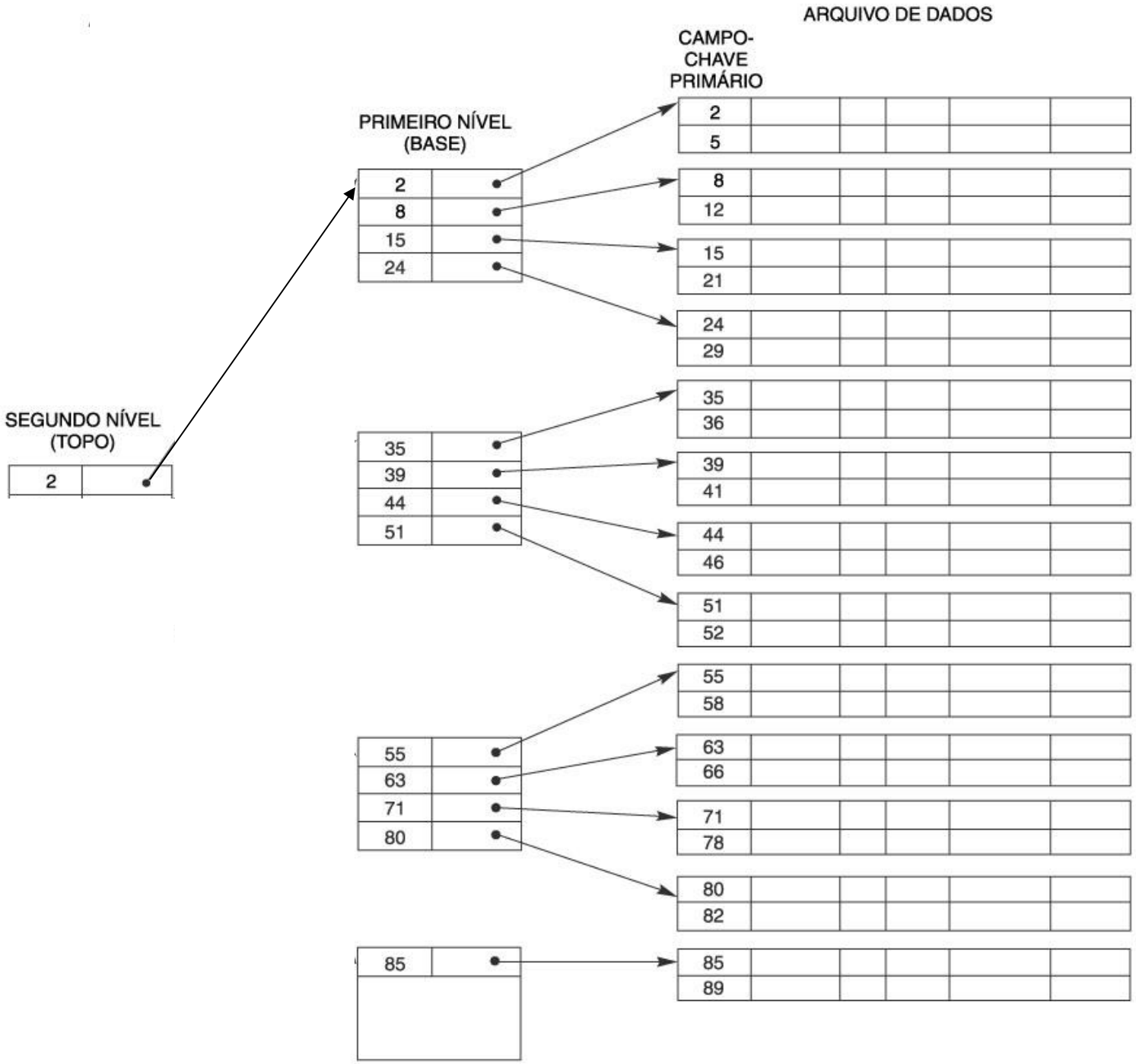
Construindo o 1º nível (Índice Primário)



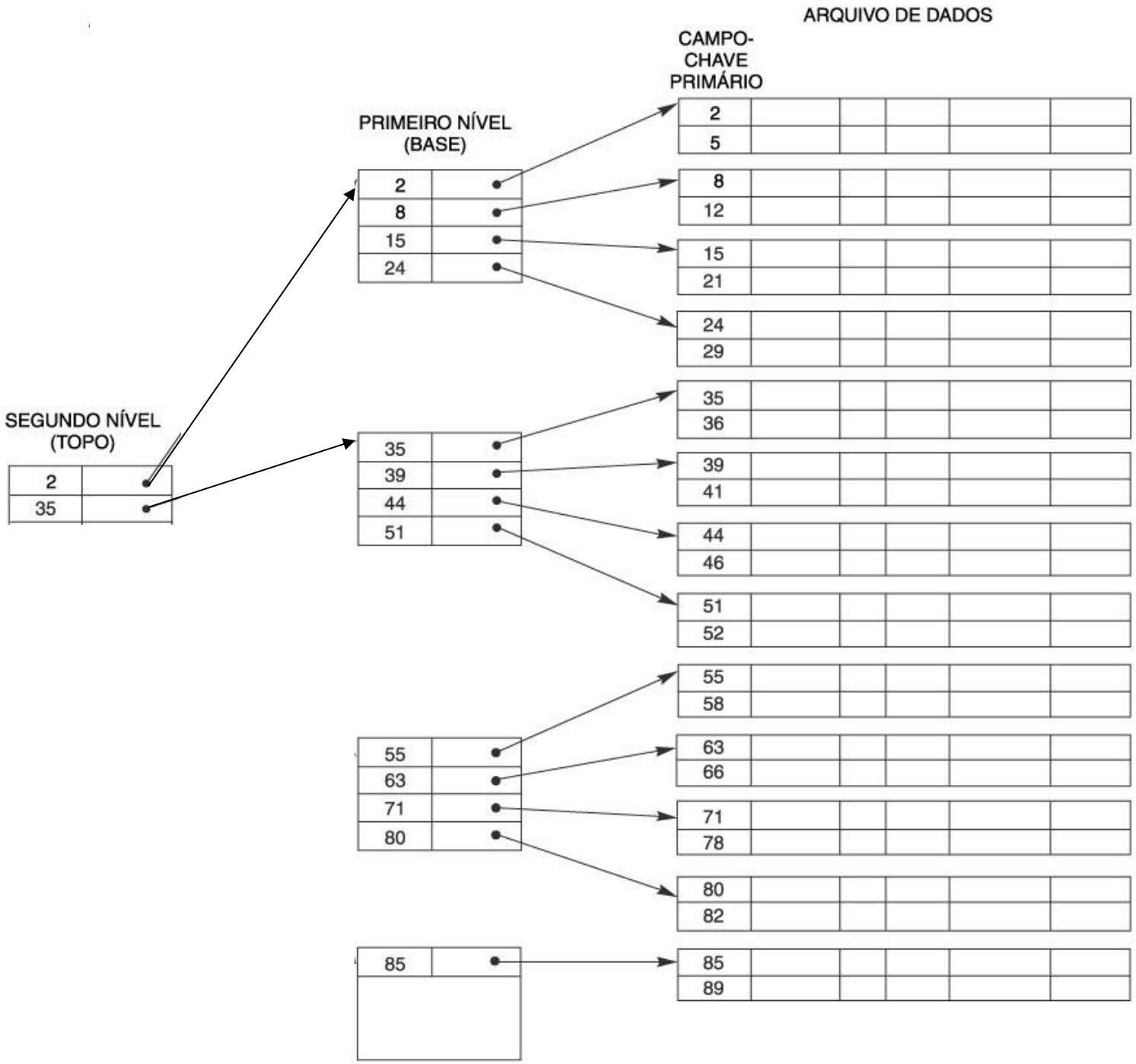
Construindo o 2º nível (Índice Primário)



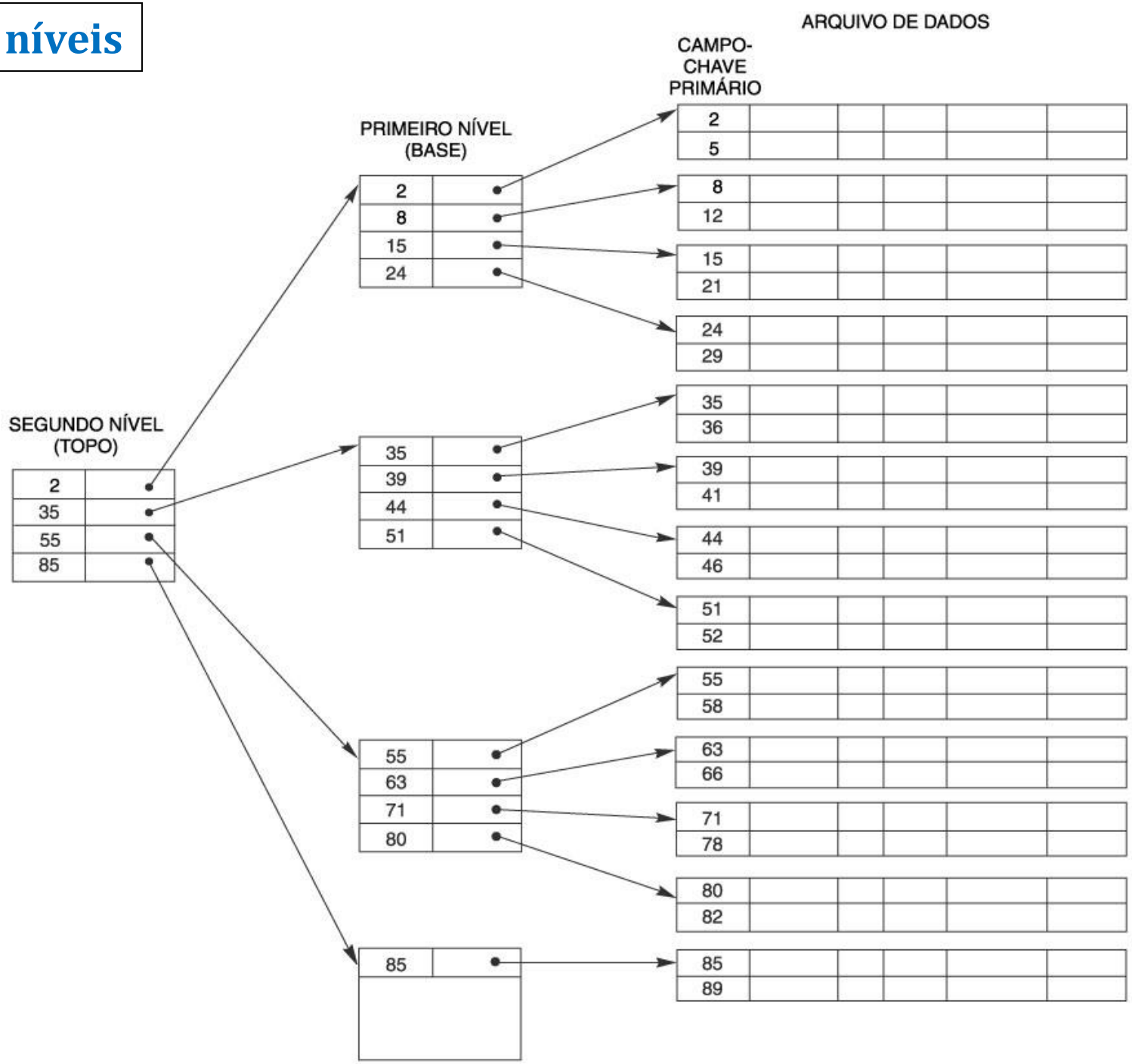
Construindo o 2º nível (Índice Primário)



Construindo o 2º nível (Índice Primário)

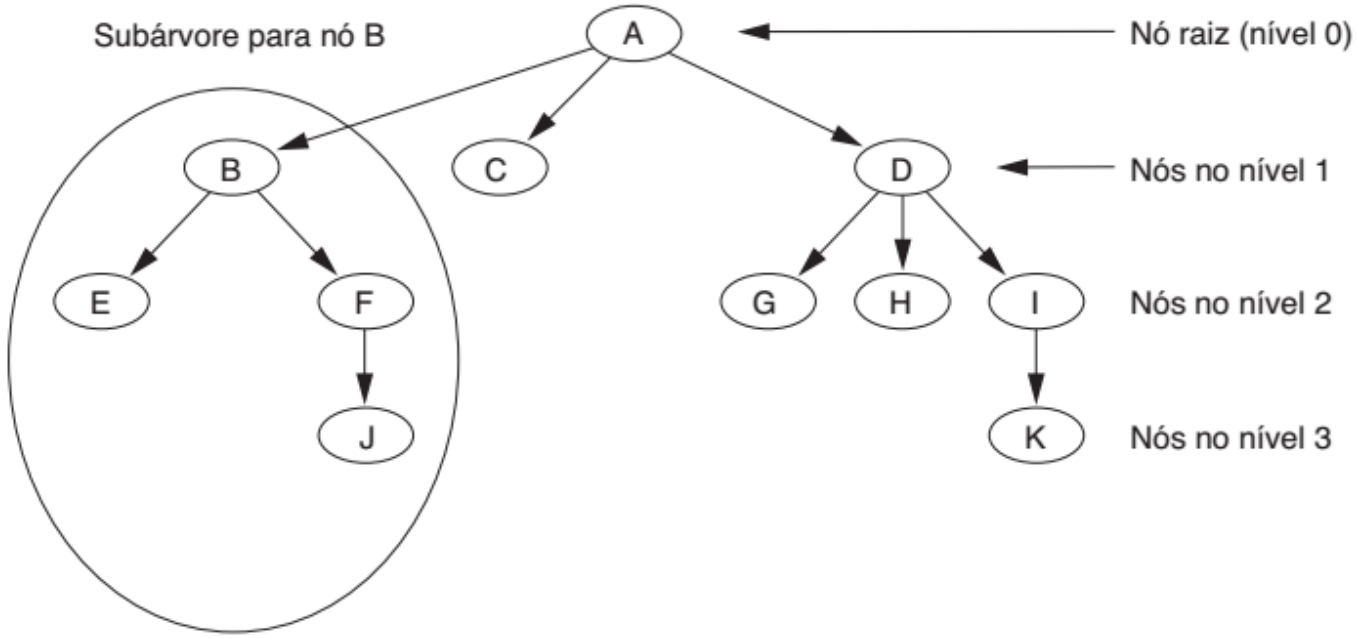


Arquivo de índice de dois níveis



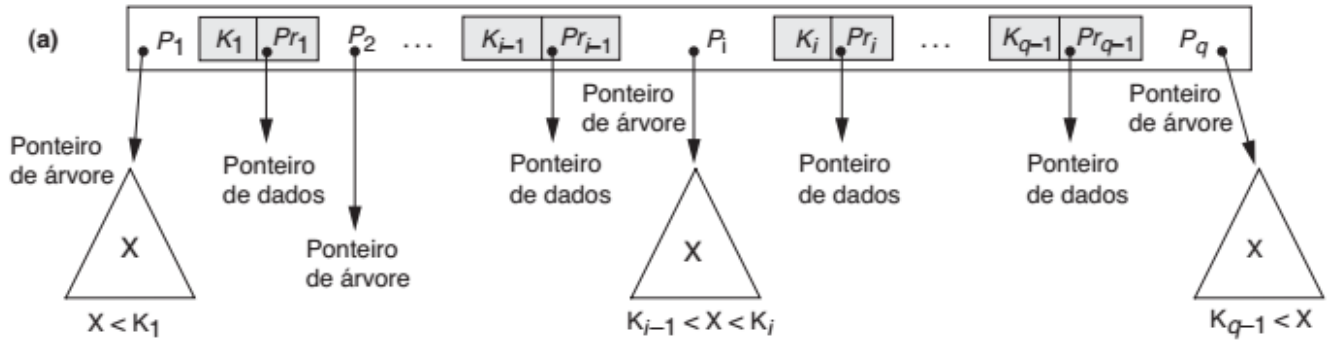
Índices multinível

- Estrutura de uma árvore
  - Uma árvore é composta de nós e um nó especial chamado **raiz** (que não tem pai)
  - Um nó que não tem filhos é denominado **folha**
  - Nós não folha são chamados de **internos**



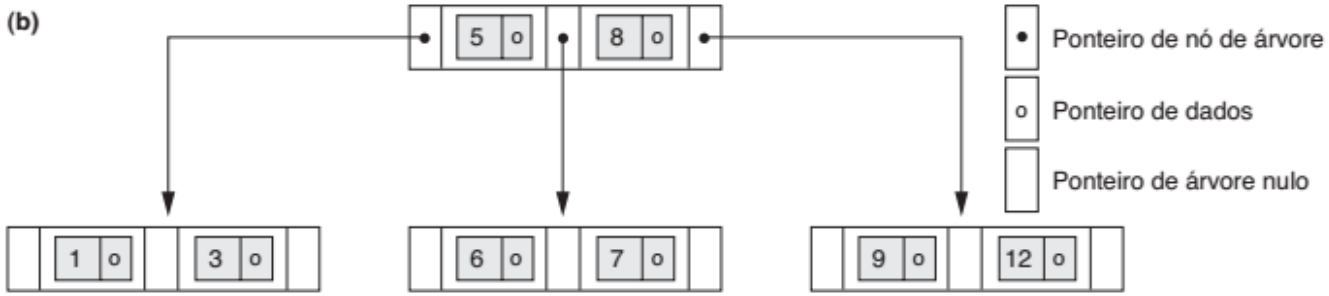
Índices multinível: B-tree

- Índice B-tree: é uma implementação das árvores B de de Lehman-Yao
  - Os índices B-tree podem tratar consultas de igualdade e de faixa, em dados que podem ser classificados em alguma ordem.



K =valor de pesquisa
P = ponteiro de dado

B-tree:
- É uma arvore balanceada
  - Os nós são distribuídos por igual
  - Velocidade da pesquisa é uniforme



INDEX

LABORATÓRIO DE BANCO DE DADOS

VANESSA BORGES

Declaração de índices em SQL

- **Existem duas maneiras de se declarar índices em SQL:**
  - **Implicitamente:** isso ocorre quando se declara uma chave primária ou candidata;
    - A restrição de integridade **PRIMARY KEY** num comando **CREATE TABLE** ou **ALTER TABLE** corresponde à definição de uma **chave primaria em SQL**.
    - A restrição de integridade **UNIQUE** corresponde à definição de uma **chave candidata** em SQL.
  - **Explicitamente:** quando se usa o comando **CREATE INDEX**.

Índices
CREATE

- Cada fabricante de SGBDs varia bastante esse comando. A sintaxe mostrada é a utilizada pelo PostgreSQL.

- O comando CREATE INDEX constrói o índice *nome_índice* na tabela especificada. Os índices são utilizados, principalmente, para melhorar o desempenho do banco de dados

- Sintaxe:

```
CREATE [UNIQUE] INDEX <nome_índice> ON
<nome_tabela>
[USING <nome_metodo>]
(<lista_de_atributos>);
```

  - Onde:
    - **UNIQUE:** faz o sistema verificar valores duplicados na tabela quando o índice é criado, se existirem dados, e toda vez que forem adicionados dados. A tentativa de inserir ou de atualizar dados que resultem em uma entrada duplicada gera um erro.
    - **USING:** O nome do método de índice a ser utilizado (exemplo: btree, hash, etc). O método padrão é btree.

Índices
ALTER

- Altera a definição de um índice

```
ALTER INDEX <nome_índice> RENAME TO <novo_nome>;
ALTER INDEX <nome_índice> OWNER TO <novo_dono>;
```

  - Exemplo:

    ALTER INDEX distribuidores **RENAME TO** fornecedores;

Índices
DROP

- Remove um índice

```
DROP INDEX [IF EXISTS]<nome_índice>;
```

- Exemplo:
  DROP INDEX idx_títulos;

Teste de índices

- Criando as tabelas para o teste:

```
CREATE TABLE testeindice (id int4);

CREATE TABLE testeindice_0 (id1 int4, id2 int4);
```

- Populando as tabelas com os dados

```
INSERT INTO testeindice SELECT * FROM generate_series(1, 10000000);

INSERT INTO testeindice_0 SELECT i AS id1, j AS id2
FROM generate_series(1, 10) i, generate_series(1, 10000) j;
```

Teste de índices - EXPLAIN

- Vamos analisar a execução dessa consulta
  - Observar o Tipo de Varredura utilizado:
    - O PostgreSQL faz uma varredura sequencial ou utiliza algum índice?
  - Avaliar o Custo de Execução:
    - Analisar o custo inicial e total estimado e verificar o tempo real de execução.
  - Examinar a Eficiência do JOIN:
    - Avaliar como o PostgreSQL executa o JOIN e se há gargalos de desempenho.

```
SELECT a.id, b.id2
FROM testeindice a
JOIN testeindice_0 b ON a.id = b.id2
WHERE a.id > 9999;
```

Índices: métodos de varredura

- **O que é um Método de Varredura?**
  - Um **método de varredura** (ou *scan*) define como o PostgreSQL acessa e lê dados de uma tabela para executar uma consulta.
  - A escolha do método depende de:
    - Existência de índices.
    - Tamanho da tabela.
    - Tipo de filtro na consulta.
  - O otimizador do PostgreSQL decide o método de varredura mais eficiente para cada consulta, considerando o custo e o tempo estimados.

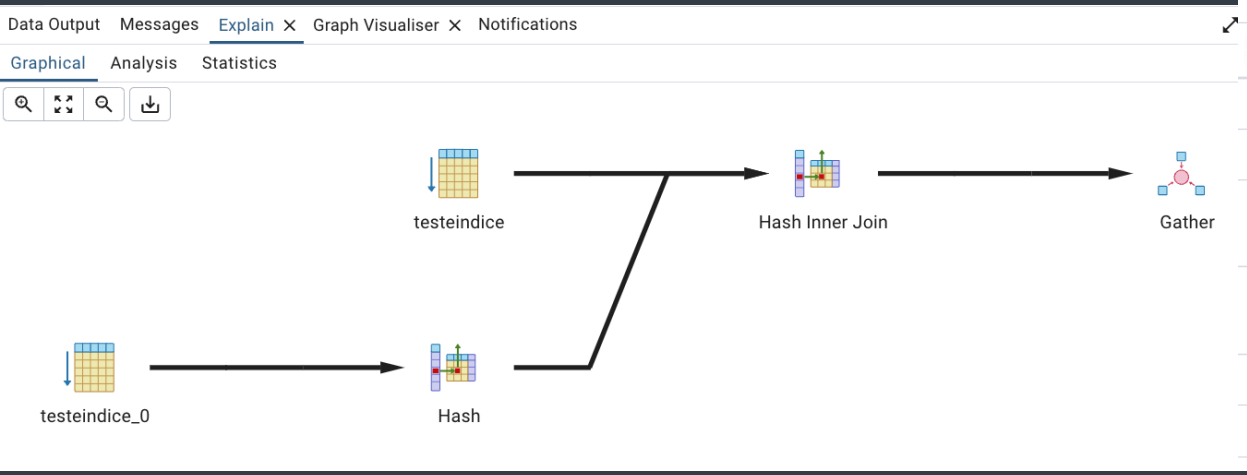
Índices: métodos de varredura no PostgreSQL

Método de Varredura	Descrição	Ideal para	Vantagens	Desvantagens
Sequential Scan (Seq Scan)	Varredura sequencial que lê todas as linhas da tabela uma por uma.	Tabelas pequenas ou quando não há índice disponível.	Simples e eficaz para tabelas pequenas.	Lento para tabelas grandes, pois lê todas as linhas.
Index Scan	Usa um índice para localizar linhas específicas e acessa a tabela para dados adicionais.	Consultas com filtros em colunas indexadas.	Eficiente para buscas rápidas em colunas indexadas.	Pode ser menos eficiente se muitos registros atenderem ao filtro.
Index Only Scan	Lê os dados diretamente do índice sem acessar a tabela.	Consultas onde todos os dados necessários estão no índice.	Muito rápido, pois evita leituras da tabela.	Funciona apenas em índices que “cobrem” a consulta.
Bitmap Index Scan	Combina vários índices usando bitmaps temporários para ler dados em blocos.	Consultas com várias condições (ex.: AND, OR).	Reduz o número de leituras e acessa dados em blocos.	Requer mais memória para armazenar bitmaps temporários.

Teste de índices - EXPLAIN

- Análise de consultas sem índice
- Para obter informações sobre a consulta usamos o comando **EXPLAIN** antes da consulta SQL

```
SELECT a.id, b.id2
FROM testeindice a
JOIN testeindice_0 b ON a.id = b.id2
WHERE a.id > 9999;
```



aula/postgres@local-postgres

No limit

Query | Query History

```
1 SELECT a.id, b.id2
2     FROM testeindice a
3     JOIN testeindice_0 b ON a.id = b.id2
4     WHERE a.id > 9999;
```

Data Output | Messages | Explain | Graph Visualiser | Notifications

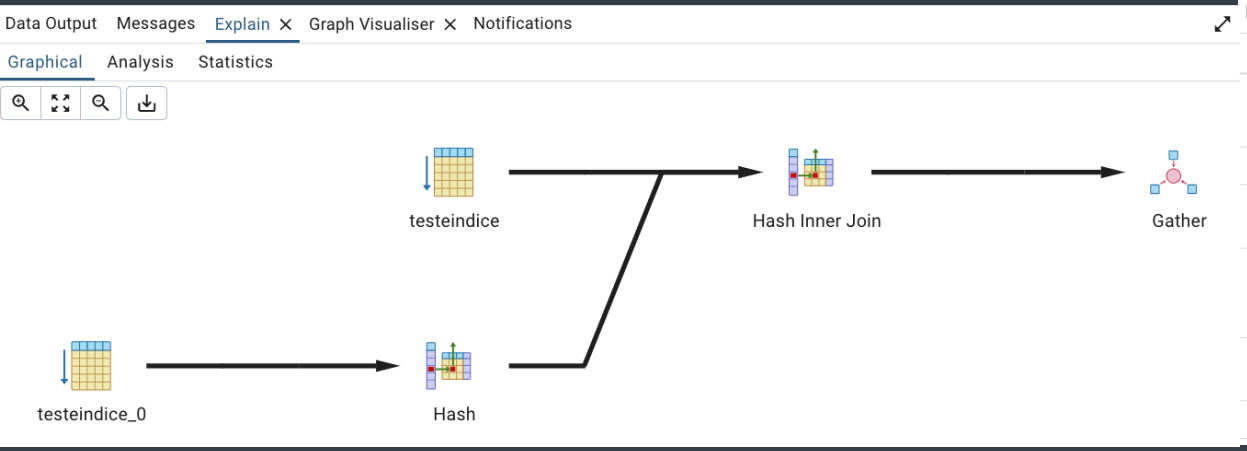
Data Output | Analysis | Statistics

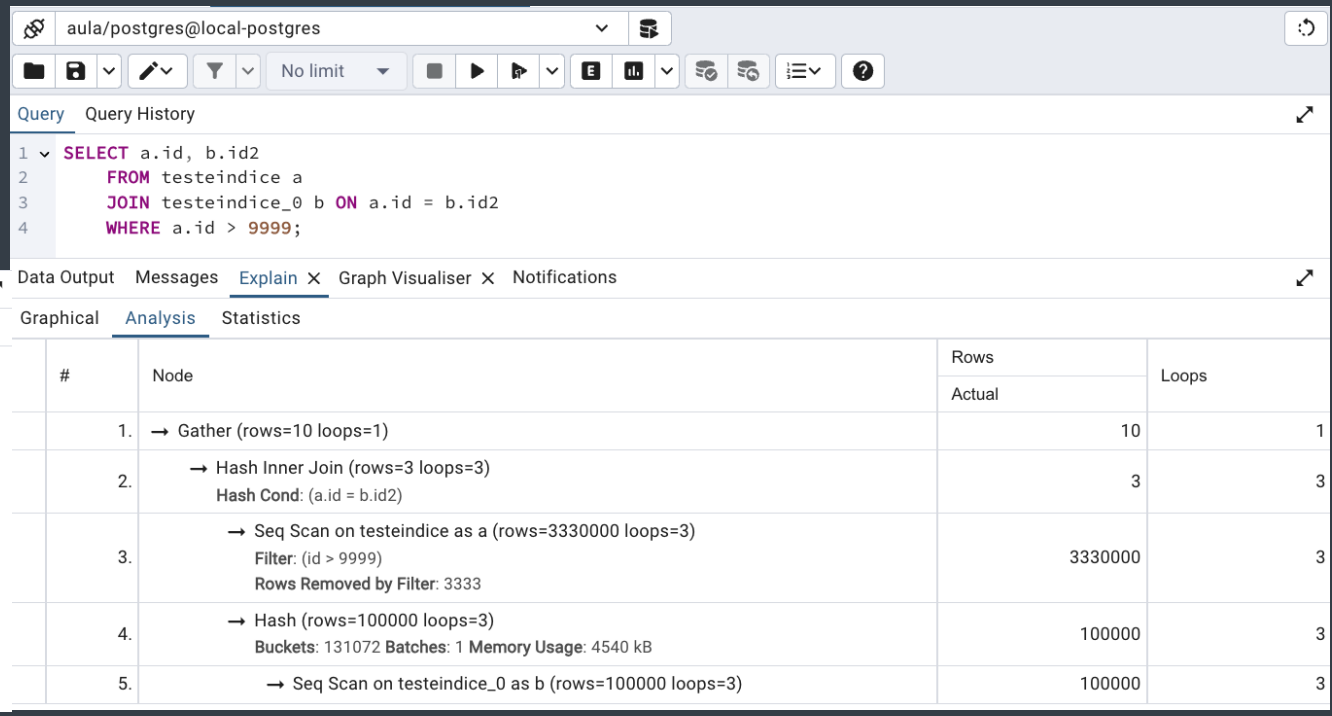
#	Node
1.	→ Gather
2.	→ Hash Inner Join **Hash Cond**: (a.id = b.id2)
3.	→ Seq Scan on testeindice as a **Filter**: (id > 9999)
4.	→ Hash
5.	→ Seq Scan on testeindice_0 as b

Teste de índices – EXPLAIN ANALYSE

- Análise de consultas sem índice
- Para obter informações sobre a consulta usamos o comando **EXPLAIN ANALYSE** antes da consulta SQL

```
SELECT a.id, b.id2
FROM testeindice a
JOIN testeindice_0 b ON a.id = b.id2
WHERE a.id > 9999;
```





#	Node	Rows Actual	Loops
1.	→ Gather (rows=10 loops=1)	10	1
2.	→ Hash Inner Join (rows=3 loops=3) **Hash Cond**: (a.id = b.id2)	3	3
3.	→ Seq Scan on testeindice as a (rows=3330000 loops=3) **Filter**: (id > 9999) **Rows Removed by Filter**: 3333	3330000	3
4.	→ Hash (rows=100000 loops=3) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 4540 kB	100000	3
5.	→ Seq Scan on testeindice_0 as b (rows=100000 loops=3)	100000	3

Principais diferenças entre EXPLAIN e EXPLAIN ANALYSE

Aspecto	EXPLAIN	EXPLAIN ANALYSE
Execução da Consulta	Não executa a consulta, apenas exibe o plano estimado.	Executa a consulta e exibe o plano com dados reais de execução.
Finalidade	Fornece um plano teórico de como o PostgreSQL pretende executar a consulta.	Fornece uma análise prática e detalhada de como a consulta realmente foi executada.
Tempo de Execução da Análise	Muito rápido, pois não executa a consulta.	Pode ser demorado, especialmente para consultas complexas, pois executa a consulta.
Tempo de Execução Real	Não exibido, apenas estimativas de custo.	Exibe o tempo real de cada etapa do plano de execução, mostrando onde o tempo foi gasto.
Número de Linhas Processadas	Exibe uma estimativa do número de linhas.	Mostra o número real de linhas processadas em cada etapa do plano.
Utilização	Útil para uma análise inicial rápida, para entender o plano estimado de execução e prever se um índice será usado.	Ideal para análise e otimização detalhada, pois identifica gargalos e confirma o impacto real de índices e outros ajustes.
Vantagem	Rapidez e utilidade para uma visão preliminar sem executar a consulta.	Precisão total, com informações detalhadas sobre o desempenho, essenciais para otimizações eficazes.

Teste de índices

- Criação de índices

```
CREATE INDEX idx_testeindice_id ON testeindice(id);
CREATE INDEX idx_testeindice_0_id1 ON testeindice_0(id1);
CREATE INDEX idx_testeindice_0_id2 ON testeindice_0(id2);
CREATE INDEX idx_testeindice_0_id1_id2 ON testeindice_0(id1, id2);
```

- Cada índice é criado para otimizar buscas específicas.
  - O índice idx_testeindice_id melhora buscas onde id é usado
  - O índice idx_testeindice_0_id1_id2 é um índice composto para consultas que filtram tanto id1 quanto id2

Índices

- **O que são Índices?**
  - **Índices** são estruturas auxiliares que ajudam a acelerar o acesso a dados em uma tabela.

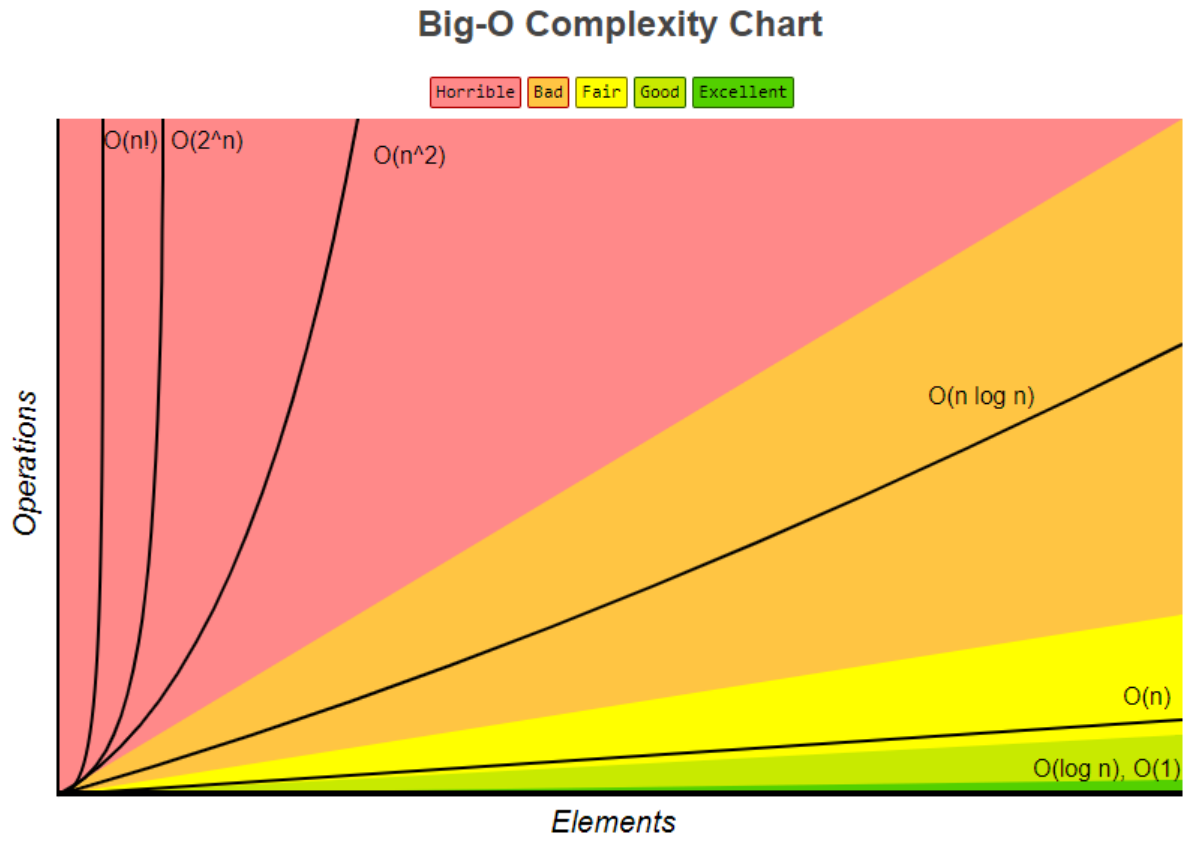
- O PostgreSQL oferece diferentes tipos de índices para otimizar buscas específicas e melhorar o desempenho da consulta.
- A escolha do tipo de índice ideal depende do tipo de consulta e das características dos dados.

Índices: tipos de índices no PostgreSQL

Tipo de Índice	Descrição	Ideal para	Complexidade	Exemplo
B-tree	Índice padrão, adequado para buscas de igualdade e intervalos.	Consultas de igualdade e intervalos (=, >, <, BETWEEN).	O(log n)	CREATE INDEX idx_nome ON tabela (coluna); CREATE INDEX idx_nome_btree ON tabela USING btree (coluna);
Hash	Usa uma função de hash para buscas rápidas de igualdade.	Consultas de igualdade (=).	O(1)	CREATE INDEX idx_nome_hash ON tabela USING hash (coluna);
GiST	Generalized Search Tree, usado para dados complexos e multidimensionais, como dados espaciais.	Dados espaciais e buscas multidimensionais.	Depende do tipo de dado	CREATE INDEX idx_nome_gist ON tabela USING gist (coluna);
GIN	Generalized Inverted Index, ideal para arrays, textos e documentos JSON.	Arrays, busca de texto, e documentos JSON.	Depende do tipo de dado	CREATE INDEX idx_nome_gin ON tabela USING gin (coluna);
BRIN	Block Range INdex, usado para tabelas grandes com dados ordenados.	Grandes tabelas com dados ordenados.	O(log n) para blocos	CREATE INDEX idx_nome_brin ON tabela USING brin (coluna);

Métodos de varredura
Índices: Hash e B-tree

- Complexidade algorítmica
  - Hash: O(1)
  - B-tree: O (logn)



Métodos de varredura
Índices: Hash e B-tree

- Complexidade algorítmica
  - Hash: O(1)
  - B-tree: O (logn)

in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX nome ON tabela USING hash (coluna);
```

Nota: Testing has shown PostgreSQL's hash indexes to perform no better than B-tree indexes, and the index size and build time for hash indexes is much worse. Furthermore, hash index operations are not presently WAL-logged, so hash indexes may need to be rebuilt with REINDEX after a database crash. For these reasons, hash index use is presently discouraged.

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the *operator class*). As an example,

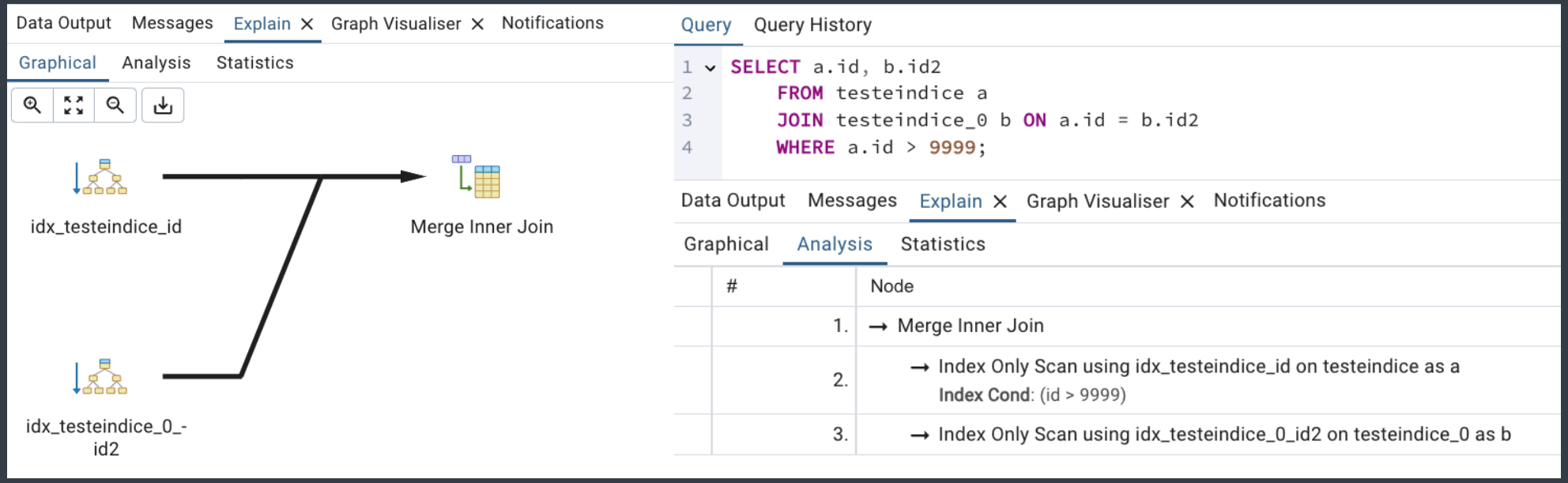
Os testes mostraram que os índices hash do PostgreSQL não têm desempenho melhor do que os índices B-tree, e que o tamanho e o tempo de construção dos índices hash são muito piores. Por estas razões, desencoraja-se a utilização dos índices hash.

http://pgdocptbr.sourceforge.net/pg82/indexes-types.html

Teste de índices
INDEXADO COM EXPLAIN

- Análise de consultas sem índice
- Para obter informações sobre a consulta usamos o comando **EXPLAIN** antes da consulta SQL

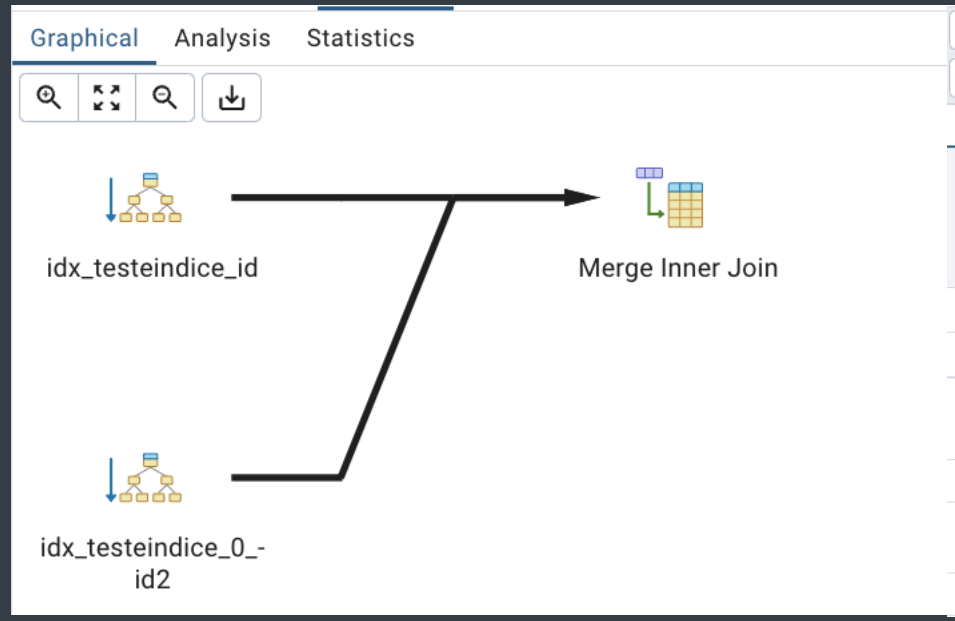
```
SELECT a.id, b.id2
FROM testeindice a
JOIN testeindice_0 b ON a.id = b.id2
WHERE a.id > 9999;
```



Teste de índices
INDEXADO COM EXPLAIN ANALYSE

- Análise de consultas sem índice
- Para obter informações sobre a consulta usamos o comando **EXPLAIN ANALYSE** antes da consulta SQL

```
SELECT a.id, b.id2
FROM testeindice a
JOIN testeindice_0 b ON a.id = b.id2
WHERE a.id > 9999;
```



aula/postgres@local-postgres

No limit

Query | Query History

```
1 SELECT a.id, b.id2
2     FROM testeindice a
3     JOIN testeindice_0 b ON a.id = b.id2
4     WHERE a.id > 9999;
```

Data Output | Messages | Explain | Graph Visualiser | Notifications

Graphical | Analysis | Statistics

#	Node	Rows (Actual)	Loops
1.	→ Merge Inner Join (rows=10 loops=1)	10	1
2.	→ Index Only Scan using idx_testeindice_id on testeindice as a (rows=2 loops=1) **Index Cond**: (id > 9999)	2	1
3.	→ Index Only Scan using idx_testeindice_0_id2 on testeindice_0 as b (rows=100000 loops=...	100000	1

Comparação EXPLAIN ANALYSE COM ÍNDICE X SEM ÍNDICE

SEM INDEX

Data Output | Messages | Explain × | Graph Visualiser × | Notifications

Graphical | Analysis | Statistics

#	Node	Rows (Actual)	Loops
1.	→ Gather (rows=10 loops=1)	10	1
2.	→ Hash Inner Join (rows=3 loops=3) **Hash Cond**: (a.id = b.id2)	3	3
3.	→ Seq Scan on testeindice as a (rows=3330000 loops=3) **Filter**: (id > 9999) **Rows Removed by Filter**: 3333	3330000	3
4.	→ Hash (rows=100000 loops=3) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 4540 kB	100000	3
5.	→ Seq Scan on testeindice_0 as b (rows=100000 loops=3)	100000	3

Tipo de Varredura: O PostgreSQL usa uma *varredura sequencial* (Seq Scan) nas tabelas testeindice e testeindice_0 porque não há índices para otimizar a busca.

COM INDEX

Data Output | Messages | Explain × | Graph Visualiser × | Notifications

Graphical | Analysis | Statistics

#	Node	Rows (Actual)	Loops
1.	→ Merge Inner Join (rows=10 loops=1)	10	1
2.	→ Index Only Scan using idx_testeindice_id on testeindice as a (rows=2 loops=1) **Index Cond**: (id > 9999)	2	1
3.	→ Index Only Scan using idx_testeindice_0_id2 on testeindice_0 as b (rows=100000 loops=...	100000	1

Tipo de Varredura: Com os índices disponíveis, o PostgreSQL usa Index Only Scan, o que significa que consegue obter os dados diretamente dos índices sem precisar acessar as tabelas.Z

Comparação EXPLAIN ANALYSE COM ÍNDICE X SEM ÍNDICE

SEM INDEX

Data Output | Messages | Explain × | Graph Visualiser × | Notifications

Graphical | Analysis | Statistics

#	Node	Rows (Actual)	Loops
1.	→ Gather (rows=10 loops=1)	10	1
2.	→ Hash Inner Join (rows=3 loops=3) **Hash Cond**: (a.id = b.id2)	3	3
3.	→ Seq Scan on testeindice as a (rows=3330000 loops=3) **Filter**: (id > 9999) **Rows Removed by Filter**: 3333	3330000	3
4.	→ Hash (rows=100000 loops=3) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 4540 kB	100000	3
5.	→ Seq Scan on testeindice_0 as b (rows=100000 loops=3)	100000	3

Hash Join: Sem um índice, o banco de dados faz um Hash Join, que é mais custoso porque requer a criação de uma estrutura de hash para combinar as colunas id e id2.

COM INDEX

Data Output | Messages | Explain × | Graph Visualiser × | Notifications

Graphical | Analysis | Statistics

#	Node	Rows (Actual)	Loops
1.	→ Merge Inner Join (rows=10 loops=1)	10	1
2.	→ Index Only Scan using idx_testeindice_id on testeindice as a (rows=2 loops=1) **Index Cond**: (id > 9999)	2	1
3.	→ Index Only Scan using idx_testeindice_0_id2 on testeindice_0 as b (rows=100000 loops=...	100000	1

Merge Join: Ao invés de Hash Join, o PostgreSQL usa um Merge Join, que é mais eficiente quando ambas as tabelas estão ordenadas pelos índices, resultando em menos tempo de execução.

Comparação EXPLAIN ANALYSE COM ÍNDICE X SEM ÍNDICE

SEM INDEX

	QUERY PLAN text
1	Gather (cost=3693.00..276954.84 rows=99914 width=8) (actual time=1059.991..1061.382 rows=10 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Hash Join (cost=2693.00..265963.44 rows=41631 width=8) (actual time=1050.744..1050.754 rows=3 loops=3)
5	Hash Cond: (a.id = b.id2)
6	-> Parallel Seq Scan on testeindice a (cost=0.00..96331.33 rows=4163070 width=4) (actual time=0.320..469.575 rows=3330000 loop...
7	Filter: (id > 9999)
8	Rows Removed by Filter: 3333
9	-> Hash (cost=1443.00..1443.00 rows=100000 width=4) (actual time=32.136..32.136 rows=100000 loops=3)
10	Buckets: 131072 Batches: 1 Memory Usage: 4540kB
11	-> Seq Scan on testeindice_0 b (cost=0.00..1443.00 rows=100000 width=4) (actual time=0.044..13.957 rows=100000 loops=3)
12	Planning Time: 0.953 ms
13	Execution Time: 1061.576 ms

O custo total estimado é alto (cost=3693.00..276954.84).

• O tempo de execução real é de aproximadamente **1061.576 ms**, que é relativamente demorado.

COM INDEX

	QUERY PLAN text
1	Merge Join (cost=120.81..3498.99 rows=99909 width=8) (actual time=17.203..17.207 rows=10 loops=1)
2	Merge Cond: (a.id = b.id2)
3	-> Index Only Scan using i_teste_id on testeindice a (cost=0.43..284433.41 rows=9990913 width=4) (actual time=0.188..0.188 rows=2 loops=1)
4	Index Cond: (id > 9999)
5	Heap Fetches: 0
6	-> Index Only Scan using i_teste_id2 on testeindice_0 b (cost=0.29..1968.29 rows=100000 width=4) (actual time=0.052..10.956 rows=100000 loop...
7	Heap Fetches: 0
8	Planning Time: 6.483 ms
9	Execution Time: 17.423 ms

O custo estimado é muito menor (cost=120.81..3498.99).

• O tempo de execução real é significativamente reduzido, com apenas **17 ms**, em comparação com os 1061 ms da consulta sem índice.

Variações de índices

- Índice único
  - Quando o índice é declarado como único, não pode existir na tabela mais de uma linha com valores indexados iguais.
    - Os valores nulos não são considerados iguais.

CREATE UNIQUE INDEX <identificador do índice>
ON <tabela> (<atributo1>, <atributo2>,...);

- Índice com vários atributos
  - Ao utilizar vários atributos num índice, o otimizador de consultas pode utilizar todas as colunas especificadas ou apenas uma ou algumas, de acordo como ele decidir.

CREATE INDEX <identificador do índice>
ON <tabela> (<atributo1>, <atributo2>,...);

Fatores que influenciam a eficiência do uso de índices

- O número de **blocos de índices** em geral é pequeno quando comparado com o número de blocos de dados;
- Tendo em vista que as chaves são classificadas, a pesquisa é rápida (pode-se usar um algoritmo de pesquisa binária);
- O índice pode ser pequeno o bastante para ser mantido permanentemente em buffers da memória principal.
  - Nesse caso, uma pesquisa para uma determinada chave envolve apenas acessos à memória principal, sem precisar de operação de I/O.